

# The Computational Theory of Intelligence: Applications to Genetic Programming and Turing Machines

Daniel Kovach

December 22, 2014

## Abstract

In this paper, we continue the efforts of the Computational Theory of Intelligence (CTI) by extending concepts to include computational processes in terms of Genetic Algorithms (GA's) and Turing Machines (TM's). Active, Passive, and Hybrid Computational Intelligence processes are also introduced and discussed. We consider the ramifications of the assumptions of CTI with regard to the qualities of reproduction and virility. Applications to Biology, Computer Science and Cyber Security are also discussed.

## 1 Introduction

In this work, we carry on the ideas proposed in [15, 16]. In [15], we posited some very basic assumptions as to what intelligence means as a mapping. Using our analysis, we formulated some axioms based on entropic analysis. In the paper that followed [16] we talked about how ideas or memes can aggregate together to form more complex constructs.

Here, we extend the concept of intelligence as a computational process to computational processes themselves, in the classic Turing sense.

The structure of this paper is as follows. In section 2, we provide deeper insights as to different types of computational intelligence, specifically in the light of executable code, section 3 the qualities of reproduction, virility, and nested programs, section 4 presents some experimental results, and sections 5 and 6 discuss applications. In the conclusion, we recapitulate and discuss future work.

## 2 Extending Concepts

Consider the definition of computational intelligence from [15]. Given the two sets  $\mathbb{S}$  and  $\mathbb{O}$ , which represent input and output sets, respectively, we introduce the mapping

$$\mathbb{I} : \mathbb{S} \rightarrow \mathbb{O}, \quad (1)$$

where  $\mathbb{I}$  essentially makes a prediction based on the input. The disparity between a prediction and reality can then be used to update  $\mathbb{I}$  in such a way that it can climb the gradient of the learning function  $\mathbb{L}$  so as to improve future performance. Although we touched upon the application of this framework to genetic algorithms [15], it will be insightful to revisit it in more detail.

## 2.1 Active, Passive, and Hybrid Computational Intelligence

Let us clarify the terminology. By *genetic algorithm* (GA), we mean a collection or *population*  $\Pi = \{\pi_i\}$  of potential solutions to a particular task, which evolves as time progresses according to some fitness function,  $\mathbb{F}$ . The function  $\mathbb{F}$  is synonymous with learning function  $\mathbb{L}$  from our previous formulation. It evaluates the quality of  $\Pi$  as a solution to the task. Specifically, if our task or *program*  $P$  is a function of  $\Pi$ , then the fitness function  $\mathbb{F}$  is nothing more than

$$\mathbb{F} = \mathbb{F}(P(\Pi)). \quad (2)$$

Note that the population  $\Pi$  is not only our input set  $\mathbb{S}$  but our output set  $\mathbb{O}$  in the representation 1. Furthermore, the mapping  $\mathbb{I}$  is nothing but a collection of mutation operations to update the  $\Pi$  at iteration  $t$  to iteration  $t + 1$ . That is

$$I : \Pi_t \rightarrow \Pi_{t+1}. \quad (3)$$

Some common mutation operations of genetic algorithms are [8]:

1. Reproduction: selecting individuals within a population and copying them without alteration into the new population.
2. Crossover: produces new offspring from elements of two other *parents*.
3. Mutation: one of the central concepts of genetic algorithms. Mutation perturbs the population or its constituents in some way. This includes altering the elements of the population directly, or changing their placement in the population in some way such as swapping their indices.
4. Editing: a manner in which more complicated functionality can be reduced to simpler algebraic rules.
5. Adding/Deleting: the ability to add a new element to a population or remove an existing element from the population.
6. Encapsulating: the process of creating a new element which represents an aggregate of functionality from one or more other elements in the population.

Some consideration should be given to the statement

$$\mathbb{I}_{t+1} = \mathbb{I}_t + \nabla \mathbb{F}_t \quad (4)$$

from [15]. For this equation to make sense, not only must the algebraic operation '+' be meaningfully defined, but the gradient operator implies that  $\mathbb{F}$  is differentiable. The gradient operation makes sense only if the domain of  $\mathbb{F}$  is a Banach space. In this context, we cannot make such a claim. Thus equation 4 becomes

$$\mathbb{I}_{t+1} = \mathbb{I}_t + \Delta \mathbb{F}, \quad (5)$$

where  $\Delta \mathbb{F} = \mathbb{F}(\Pi_t) - \mathbb{F}(\Pi_{t-1})$ . Observe that although the mathematical formulation is different, the intent still holds. The '+' operator communicates the number of permutations proportional to  $\Delta \mathbb{F}$ .

Since the information is retained in the population, and only improved through evolutionary perturbations, we consider this an example of *passive computational intelligence*. This stands in contrast to *active computational intelligence*, where we are updating the mapping  $\mathbb{I}$  itself. Any combination of the two methods can be considered a form of *hybrid computational intelligence*.

## 2.2 Genetic Algorithms

Recall the entropic considerations from [15], where we specified that

- *Computational intelligence is a process that locally minimizes and globally maximizes entropy.*

Since the population  $\Pi$  is the output set, it is the entropy of the population that must be minimized by the intelligence process. Equivalently, we can say that as the amount of iterations becomes sufficiently large, the difference between  $\Pi_t$  and  $\Pi_{t-1}$  becomes increasingly negligible. In other words, assuming the subtraction operation is meaningfully defined, we have

$$\lim_{t \rightarrow \infty} \Pi_{t+1} - \Pi_t = 0. \quad (6)$$

This makes sense, for we expect that as time increases, the fitness of our solution reaches an optimum, so that eventually there is no need to update. In other words, our population becomes more consistent over time.

Of course, the typical considerations regarding genetic algorithms such the necessity of avoiding local optima, the proper choice of a fitness function, the complexity of the algorithm itself, among others, still hold [8].

As for global entropy considerations, we must consider the role of the population in its realization under its program  $P$ . As mentioned before, the total entropy of a system can be written as:

$$S = \sum_{i=1}^N s_i \quad (7)$$

where  $\{s_i\}$  represent each respective source of entropy. In particular, under the considerations of this section, we have

$$S = s_{\Pi} + s_P, \quad (8)$$

where  $s_{\Pi}$  and  $s_P$  represent the entropy due to the population itself and the entropy from the program  $P$ , respectively.

To calculate the entropy, we used the Renyi entropy of order  $\alpha$ , where  $\alpha \geq 0$  and  $\alpha \neq 1$  defined as

$$\mathbb{H}_{\alpha}(X) = \frac{1}{1-\alpha} \log \left[ \sum_{i=1}^N \mathbb{P}[x_i]^{\alpha} \right]. \quad (9)$$

where  $\mathbb{P}[x_i]$  is the probability associated with state  $x_i \in X$ .

## 2.3 Turing Machines

Let us consider the discussions of the previous sections in terms of code that is executable in the Turing sense. We will apply the formulations we have considered thus far and make improvements to refine the theory in this light.

Recall that a Turing machine  $M$  is a hypothetical device that manipulates symbols on a tape, called a Turing tape, according to a set of rules. Despite its simplicity, the Turing machine analogy can be used to describe certain aspects of modern computers [21], and more [13].

The Turing tape has some important characteristics. The tape contents, called the *alphabet*, encode a given table of rules. Using these rules the machine manipulates the tape. It can write and read to and from the tape, change the configuration, or *state*  $\Sigma$  of the machine, and move the tape back and forth [11].

Typically, the instructions on the Turing tape consist of at least one instruction that tells the machine to begin computation, or a START instruction, and one to cease computation, called a STOP or HALT instruction [11]. Other instructions might include MOVE, COPY, etc, or even instructions that encode no operation at all. Such instructions are commonly called NOOP's (NO OPERATION). The total number of instructions depends on the situation and the application at hand, but for the purposes of this paper, we will consider only tape of finite length for which the instruction set is also finite and contains at least the START and STOP instructions. In fact, for our purposes, the minimum amount of opcodes a program needs to run, or to be *executable*, are the START and STOP instructions.

Now, consider the set of all instructions available to a particular  $M$ . To be consistent with our notation thus far, we will denote this set as  $\tilde{\pi}$ , where

$$\tilde{\pi} = \{\pi_0 = \text{START}, \dots, \pi_n = \text{STOP}\} \quad (10)$$

From this set, a particular sequence of instructions  $\Pi$  can be formed. Although many synonymous terms abound for this sequence of instructions, including a code segment, executable, or tape, we will refer to it simply as the

*code.* We consider the execution or implementation  $\Psi$  to be the execution of the code on the Turing machine  $M$ , which produces could produce another code segment, and change the machine state. Expressed in notation:

$$\Psi : \tilde{\Pi} \times \tilde{\Sigma} \rightarrow \tilde{\Pi} \times \tilde{\Sigma}. \quad (11)$$

where  $\tilde{\Pi}$  and  $\tilde{\Sigma}$  represent the set of all code and machine states, respectively. For the majority of this paper, it is the code that is of primary concern, and thus we can omit the machine state to simplify the notation when this context is understood:

$$\Psi : \tilde{\Pi} \rightarrow \tilde{\Pi} \quad (12)$$

Our fitness function  $\mathbb{F}$  is a measure the efficaciousness of the implementation of our program by some predefined standard. Later, we will present some specific examples used in application.

We are still free to think of the intelligence mapping  $\mathbb{I}$  as a perturbation mapping as we did in section 2.1. The only difference is the physical interpretation of the term 'population'. This case concerns machine code instructions. As with genetic algorithms, perturbing code may advance or impede the fitness of the program. In fact, it may even destroy the ability to execute at all, if one either the START or STOP operations are removed.

A final note should be made as to the entropic considerations of the Turing Machine. In this light, we must take into account all the workings of the machine and thus 8 becomes

$$S = s_{\Pi} + s_M \quad (13)$$

where  $s_M$  is the entropy incurred by the Turing Machine, which includes its state  $\Sigma$ .

### 3 Reproduction, Viruses, and Nested Programs

In this section, we will continue the above analysis, and discussed some qualities that emerge based on our assumption.

#### 3.1 Formalisms of Code and Other Qualities

We can apply set theoretic formalisms in the usual sense. For example, consider a code  $\Pi$  and an instruction,  $\pi \in \tilde{\Pi}$ . Then the notation  $\pi \in \Pi$  communicates the fact that  $\pi$  is an instruction in  $\Pi$ . Further, set theoretic operations follow naturally. Let  $\Pi_1, \Pi_2 \in \tilde{\Pi}$ . In the case of the union operation, we have

$$\Pi_1 \cup \Pi_2 = \{\Pi_1, \Pi_2\} \quad (14)$$

which is a set containing the two programs  $\Pi_1$  and  $\Pi_2$ . Similarly, with the intersection operation, we have

$$c \in \Pi_1 \cap \Pi_2 \Leftrightarrow c \in \Pi_1 \wedge c \in \Pi_2, \quad (15)$$

or the set of all contiguous code segments  $c = \{\pi\}$  that are contained by both  $\Pi_1$  and  $\Pi_2$ . Note that this code segment need not be executable. Also, if it is of particular utility in the program it may be *degenerate* in that it may appear multiple times throughout the code.

For set inclusion, we say that

$$\Pi_1 \subset \Pi_2 \Leftrightarrow \pi \in \Pi_2 \forall \pi \in \Pi_1 \quad (16)$$

Proper containment evolves naturally if  $\Pi_1 = \Pi_2$ .

### 3.2 Metrics and Neighborhoods

The differences between code and real numbers or sets in conventional mathematics make it at first difficult to form a distance metric  $d$  on them. Still, some have been explored such as the Levenshtein distance [9], Damerau-Levenshtein distance [2], Hamming distance [10], and the Jaro-Winkler distance [12].

Given a metric  $d$ , we can form an open ball [7], or neighborhood  $B$  centered at some code  $\Pi_0$  consisting of all programs  $\Pi$  such that for some real number  $\epsilon$ ,  $d(\Pi_0, \Pi) < \epsilon$ . In other words

$$B(\Pi_0, \epsilon) = \{\Pi : d(\Pi_0, \Pi) < \epsilon\}. \quad (17)$$

We can utilize the developments above to relax the definition of set inclusion with respect to the metric,  $d$ . Specifically, define

$$\Pi_1 \overset{\epsilon}{\subset} \Pi_2 \Leftrightarrow d(\Pi_1, \Pi) < \epsilon, \Pi \subset \Pi_2$$

Of course, similar considerations can be extended to proper containment. Such code segments that lie suitably close to each other with respect to  $d$  will be referred to as *polymorphic* with respect to  $d$ .

### 3.3 Reproduction

Consider the mapping

$$\Psi_i(\Pi) = \Pi' \quad (18)$$

where  $\Pi'$  is in the neighborhood of  $\Pi$ . Of particular interest is when  $\Pi'$  not only lies within the neighborhood of  $\Pi$  but produces a machine state  $\Sigma'$  suitably close to that of  $\Pi$ , as valued by some meaningful metric. We note in particular the condition when  $\Pi'$  also encodes for reproduction. Unless in the presence of polymorphic programs [20], we will assume the code copies itself exactly, and replace  $\Pi'$  with  $\Pi$  in notation.

Sequential program execution will yield the same result

$$\Psi_t = \Psi(\Psi_{t-1}), \Psi_0 = \Psi(\Pi) \quad (19)$$

where

$$\Psi_i(\Pi) = \Pi, \forall i. \quad (20)$$

Suppose that a particular tape is able to produce, say  $N$  copies of itself. Then the entropy of the system is simply

$$S = s_M + \sum_{i=0}^N s_{\Pi_i}, \quad (21)$$

where  $s_0$  refers to the original copy of the tape. By comparison with equation 13, the entropy is proportional to the number of progeny in addition to the machine state.

### 3.4 Viruses

Thus far, we have been considering only predefined programs. But a well known concept from nature concerns when external code is injected into a host program, or otherwise hijacks the computational resources of the system. One key characteristic of viruses is their ability to proliferate. Consider  $\nu$  such that

$$\nu \in \Psi(\Pi), \nu \subseteq \Pi \quad (22)$$

If the viral code,  $\nu$  is executable, we will call the virus  $\nu$ -executable. If the viral code also can reproduce, that is  $\nu \in \Psi(\nu)$ , we will call this quality  $\nu$ -reproductive. Finally, it is often the case that viral code is not reproductive or executable. Its task is to merely copy or install some payload [18], here denoted by  $\rho$ . In this case,

$$\rho \in \Psi(\Pi), \rho \subseteq \nu \subseteq \Pi \quad (23)$$

Viral code most often is pernicious, though this is not necessarily the case. For example, up to 8% of the human genome derives from retroviruses [3]. To determine the nature of the effect of the virus, we look at the change in fitness before and after its influence.

The influence of viral code on program fitness falls into three categories. Consider  $\Pi, \nu \cap \Pi = \emptyset$ , and  $\Pi', \nu \cap \Pi' = \nu, \Pi = \Pi' - \nu$ . In other words,  $\Pi'$  code in the presence of a virus, from the otherwise pristine  $\Pi$ . The three cases are  $\mathbb{F}(\Pi') - \mathbb{F}(\Pi) > 0$ ,  $\mathbb{F}(\Pi') - \mathbb{F}(\Pi) = 0$ , and  $\mathbb{F}(\Pi') - \mathbb{F}(\Pi) < 0$ . These cases are of enough significance that they deserve their own nomenclature. We will refer to these cases as commensalistic, symbiotic, and parasitic, respectively, out of consistency with their counterparts from biological sciences.

### 3.5 Nested Programs

One final case to be considered is when a program or multiple programs can contribute to produce aggregate programs of greater complexity. In such a case, the program, when executed, would not only alter machine state and produce

code as in 11 but will also contain a product code segment, and product code machine state,

$$\Psi : \tilde{\Pi} \times \tilde{\Sigma} \rightarrow \tilde{\Pi} \times \tilde{\Sigma} \times \tilde{\Pi}^1 \times \tilde{\Sigma}^1 \quad (24)$$

Introduce the program

$$\Psi^1 = \tilde{\Pi}^1 \times \tilde{\Sigma}^1 \rightarrow \tilde{\Pi}^1 \times \tilde{\Sigma}^1 \quad (25)$$

Of course, we could nest this process and define

$$\Psi^1 = \tilde{\Pi}^1 \times \tilde{\Sigma}^1 \rightarrow \tilde{\Pi}^1 \times \tilde{\Sigma}^1 \times \tilde{\Pi}^2 \times \tilde{\Sigma}^2, \quad (26)$$

with  $\Psi$  from 24 denoted as  $\Psi^0$ , we could write

$$\Psi^n[\Psi^{n-1}[\dots\Psi^0[\Pi^0 \times \Sigma^0]\dots]] \rightarrow \tilde{\Pi}^n \times \tilde{\Sigma}^n \quad (27)$$

where the superscript indicates the level of nesting of the program.

Just as we discussed programs producing multiple progeny, we can also let them produce multiple product code. We will denote each copy with a subscript  $i_m = \{0, 1, \dots, I_m\}$ . Thus the above becomes

$$\Psi^n[\Psi_{i_{n-1}}^{n-1}[\dots\Psi_{i_0}^0[\Pi_{i_0}^0 \times \Sigma_{i_0}^0]\dots]] \rightarrow \tilde{\Pi}_{i_n}^n \times \tilde{\Sigma}_{i_n}^n. \quad (28)$$

To determine the total entropy, simply sum over all product code.

$$S = s_M + \sum_{j=0}^{I_k} \sum_{k=0}^n \tilde{\Pi}_j^k \times \tilde{\Sigma}_j^k \quad (29)$$

This time we see that the entropy of the code varies exponentially with product and copy code.

## 4 Experimentation

Consider the following scenario. We have a set of instructions, and a construct for a tape that encodes them. In the following experiments, the code is created randomly. We apply a GA to the code to achieve interesting results. The code involved had no write back capabilities, and thus the experiment was purely passive.

### 4.1 Genetics and Codons

A major focus for future work is the application of this research to computational molecular biology. Hence, we chose to implement the instruction set that nature has chosen for genetics in the following experiments.

For the base instruction alphabet, we will choose that of RNA [19]. This instruction set is composed of the 'bits' Adenine (A), Uracil (U), Guanine (G),



and Thymine (T). Each instruction can be represented as three of these quaternary bits, for a total of 64 possible outcomes. We have chosen this quaternary codon formulation to be the basis of our tape.

In both experiments, the choice of operational codes, or *opcodes*, are taken directly from these 64 quaternary bit codons. The choice to represent an opcode by a particular codon is completely arbitrary. All codons appear with the same probability in experiment, with the exception of the STOP opcode, as we will see in section 4.2.

We present two instructions sets, and two different experiments. In the first experiment, we determine how many iterations our genetic algorithm takes to produce code that is executable, and code that reproduces itself. In the second experiment, we compare the amount of reproductions of code with the total entropy incurred by the simulation.

## 4.2 First Instruction Set

For the first experiment we constructed a robust instruction set consisting of the following:

1. START (AAA): Commences program execution.
2. STOP (AUA, ATC, ATG): Halts program execution. Observe that there are three STOP codons, but only one START codon. This mirrors the amount of codons in RNA and DNA observed in nature [4].
3. BUILD\_FR (CUC): Copies to product code starting from the next respective instruction to that which comes before the BUILD\_TO (GCG) instruction.
4. COND (UUC, UUA, GAA): Sets an internal variable in the Turing machine state called a *flag*. A flag can be thought of as an internal Boolean variable. Every time one of the COND opcodes is encountered the sign of the flag is switched, that is  $\text{COND} = \neg\text{COND}$ .
5. IF (AAU): Only executes the following instruction if the COND flag has been set.
6. COPY\_ALL (AAG): Copies the entire tape to progeny.
7. COPY\_FR (CCC): Copies to progeny code starting from the next respective instruction to that which comes before the COPY\_TO (GGG) instruction.
8. JUMP\_FAR\_FR, (CUU) and JUMP\_NEAR\_FR, (AGA): both require a JUMP\_TO, (CAC, GUG) instruction. The JUMP family of instructions is designed to continue program execution at the index of the JUMP\_TO instruction. The former will find the JUMP\_TO address that is farthest away, while the JUMP\_NEAR finds the closest JUMP\_TO instructions. If there is no multiplicity in the JUMP\_TO instruction, then JUMP\_FAR\_FR, (CUU) and JUMP\_NEAR\_FR will produce the same result.

9. REM\_FR, (GCU) and REM\_TO, (UAA): Removes all code in between these instructions.

### 4.3 Second Instruction Set

Although the code of the previous section was robust, it was redundant. For example, the functionality of the code to copy itself to progeny should be a desirable trait of the *code itself*, not just an opcode that accomplishes the task in one step. Further, the remove operations can be handled via mutation operations (albeit far less efficiently) in the genetic algorithm itself.

Note that in Experiment 1, most operations have a partner or *conjugate* opcode. The necessity of conjugate pairs of instructions follows from the fact that two pieces of information are required to complete the instruction, as shown below:

```
[ instruction ] ... < code > ... [ instruction dual ]
```

We will call such instructions *dual*. The necessity for a predefined conjugate opcode can be mitigated by using the concept of *addressing*. Instead of conjugate opcodes, the instruction immediately following the dual can be thought of as an argument. The machine uses this address (unless the address represents opcode) as a manner of locating the conjugate to the dual and thus reducing the instruction set. Specifically:

```
[ instruction ] [ address ] ... < code > ... [ address ]
```

Under these considerations (and omitting the ability to build to a product), we have need for only six instructions: START, STOP, IF, COND, COPY, and JUMP. The first four instructions are exactly the same as in Experiment 1. The COPY instruction concatenates section of the tape to progeny, and the JUMP instruction continues program execution at a specified address. These final two are conjugates, the duals of which are handled via the addressing method entailed above. Notice although the lexicon is different, they have the same functionality as their respective counterparts in the first instruction set.

### 4.4 Experiment 1: Execution and Reproduction

The goal of the first instruction set was to produce code that was executable, and code that could reproduce. The code was perturbed by a GA randomly. That is, no fitness function was used in the perturbation of the code strings. Each epoch of the simulation terminated once an executable or reproducible code was discovered. For each experiment, a maximum of 1,000,000 computational iterations were allowed.

The results of the first experiment using the first instruction set are shown in table 1. The results of the first experiment using the second instruction set are summarized in table 2. Observe that the number iterations in the first instruction set was much less than that of the second, which is likely due to the disparity in robustness between the two. Also note how wildly the results vary

Table 1: Results: Experiment 1, Instruction Set 1

Experiment	Average	Std. Dev	Experiments
Executable Code	184.083	126.202	100,000
Reproductive Code	330.964	243.924	96,964

Table 2: Results: Experiment 1, Instruction Set 2

Experiment	Average	Std. Dev	Experiments
Executable Code	283.387	210.359	1,000,000
Reproductive Code	2003.24	96504.8	1,000,000

especially in terms of the reproductive capabilities of the second instruction set as demonstrated by the large standard deviation in results.

## 4.5 Experiment 2: Reproduction and Entropy

Upon answering the basic questions like the amount of computational iterations necessary to produce executable and reproductive code, the next experiment focused on the total entropy incurred by a given code segment and its offspring. Here, recall that we are considering the entropy produced by not only a given tape, but its progeny as well. We will continue with the alphabets of Experiment 1 and 2, respectively.

We bounded the total computational iterations per experimental run at 1,000,000. The maximum progeny allowed was 50. Mutations were applied by the GA according to a fitness function which evaluated the quality of the code with respect to its Renyi Entropy (of order  $\alpha = 2$ ).

The results are summarized in table 3. Observe that the entropy and the amount of reproductions appear to be correlated as demonstrated by the r-values in table 3 and in the correlation maps in Figure 1. Note that some programs appear to have achieved reproductive qualities without substantially increasing entropy but the converse is not true.

Although the actual data summarized in table 3 was one of the primary goals of this experiment, the operational characteristics of the code itself were also of interest. The graphics in figure 2 represent some interesting code behaviors. In the graphs below, each opcode was represented by a number in the following manner: START = 0, COPY = 1, JUMP = 2, IF = 3, COND = 4, STOP = 5. With these numeric values we can visualize program behavior graphically.

Although a myriad of behaviors were observed, in Figure 1 we have selected some graphs of the different behaviors. Of specific interest were the non-terminating programs. Each of these selections also exhibited reproductive capabilities. Observe that although towards the beginning of program execution, a variety of instructions were executed, eventually the programs under consideration converged to some sort of periodic behavior, a tendency we will call *asymptotic periodicity*.

Table 3: Results: Reproduction vs. Entropy

	Alphabet 1	Alphabet 2
Total Simulations	1,025,043	577,876
Average in Reproductions	17.1433	39.3921
Std. Dev. in Reproductions	23.7111	20.1516
Average in Entropy	12.4556	34.8463
Std. Dev. in Entropy	18.5525	17.6445
r-value	0.870675	0.983243

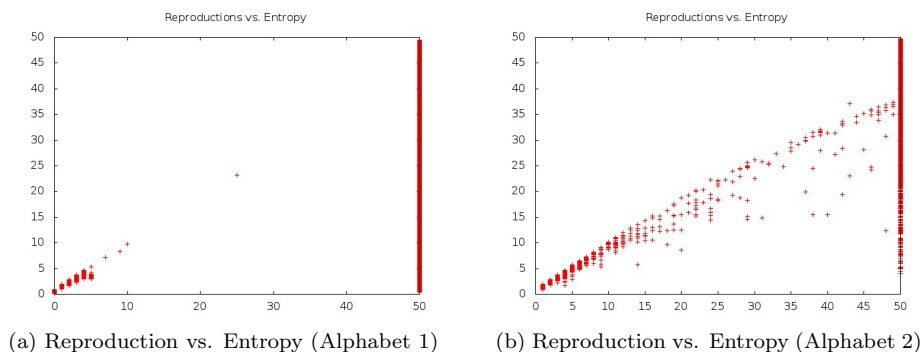


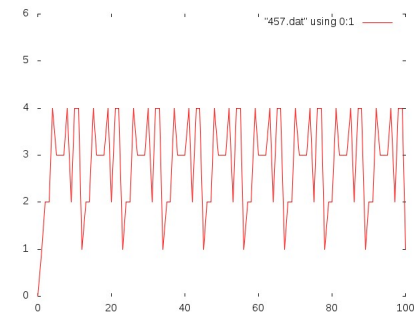
Figure 1: Graphs of reproductive iterations plotted against the total entropy of the simulation, for the first and second instruction sets, respectively.

The graphs selected not only demonstrated asymptotic periodicity but reproduced continuously in the sense that the progeny produced reached the maximum allowable threshold.

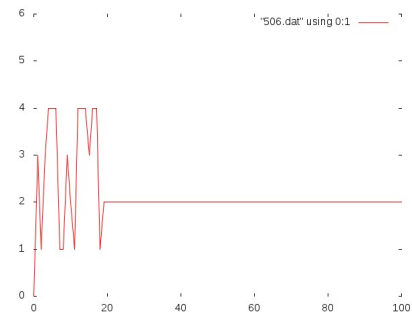
## 5 Applications to Biology

Readers familiar with genetics will notice a strong correlation between our computational formulations and genetics. In fact, genetics has been an inspiration for this research, specifically with regard to passive and hybrid forms of CTI. Recent research has been devoted to the crossover between genetics and computer science. In fact, it has been demonstrated in [13] that DNA computing is Turing complete. This has grand implications for the very nature of the definition of life itself.

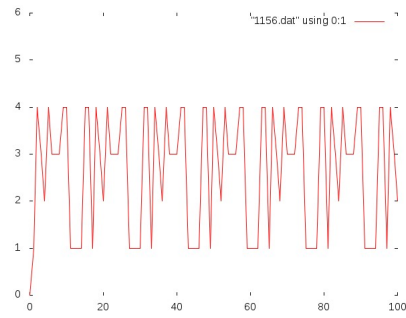
The definition of life has always been somewhat controversial. In [1], life is defined as a characteristic that distinguishes objects with self sustaining processes from those that do not. From the vantage point of Biology, life is defined as a characteristic of organisms that exhibit all or most of the following characteristics: homeostasis, organization, metabolism, growth, adaptation, response



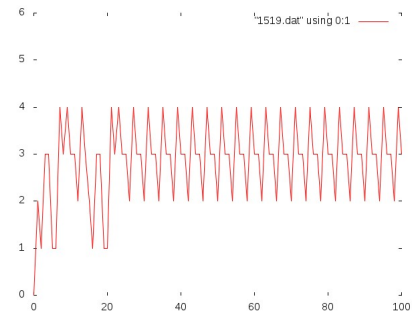
(a) Program Behavior from Sample 457



(b) Program Behavior from Sample 506



(c) Program Behavior from Sample 1156



(d) Program Behavior from Sample 1519

Figure 2: Four examples of asymptotically periodic code behaviors taken from samples of code that reproduced continuously in experiment 2.

to stimuli, and reproduction [14]. Many sources stress the chemical nature of life, and mandate that living things contain Carbon, Hydrogen, Nitrogen, Oxygen, Phosphorous and Sulfur [17].

In this paper, we introduce a definition for life as a self sustaining passive or hybrid CTI process. Embedded in this definition are well known assumptions regarding qualities one commonly associates with living things. It is known that the purpose of life is to survive, a fact that is redundant in the light of CTI. The fact that life is fitness maximizing, and increases entropy follows directly from the framework of CTI as well. Homeostasis follows naturally from the entropy minimizing portion of the fundamental axiom of CTI. If an organism is in homeostasis, this implies that the internal state space is ordered and transitions can be predicted with high probability which implies a lower entropy. Further, note the similarities with asymptotic periodicity from the samples shown in 2. Of course, no discussion about life would be complete without mentioning reproduction, which we showed in this paper follows as a natural consequence of CTI.

As far as we know, Earth is the only planet in the universe known to sustain life. However, recent advances in planetary sciences have discovered the possibility of habitable zones in other solar systems. This coupled with the observed robustness of life on Earth in extreme environments, has raised interest the application of biological concepts to the search for life beyond the confines of Earth. The vastness of these environments forces us to rethink and generalize what we know as life itself. One fantastic example of such an esoteric formulation of life is the discovery of arsenic based life right here on Earth. In December of 2010, NASA unveiled a discovery of a bacterium that integrated Arsenic into its cellular structure and even its DNA. They found that the bacterium was able to substitute arsenic for phosphorous, one of the six essential ingredients of life. This challenges conventional notions of life and seems to encourage a more general definition. In this paper, we advocate a *computational* approach to the definition of life over a chemical one.

## 6 Applications to Computer Science and Cyber Security

Applications to cyber security abound as well. In the past, computer viruses typically served little more than the entertainment of the author. Now, malicious software, or *malware*, is a commoditized industry. Patrons will hire teams of programmers to create specialized malware for end goals such as espionage, ransomware, or identity theft. As such, mitigating these threats has become a priority in most industries and even an interest in national security. For example, the US Defense Advanced Research Projects Agency (DARPA) has issued a 'Cyber Grand Challenge' offering up to \$2 million for designs of a fully automated software defense system, capable of mitigating software exploitation in real time [5]. This is just one example of increased efforts in this area. In fact,

the US president is proposing to increase the cyber defense budget to \$14 billion in 2014, more than a billion dollar increase from that of the previous year [6].

How might this research be of merit in this field? Currently, most anti-malware software (AM) stores hashes of actual malware samples against which to compare new potential threats. This has multiple disadvantages in that it ties users to a provider which may be an issue if internet connectivity is a problem. Frequently such hashes fail with polymorphic code.

Recently, much research has been conducted looking at a behavioral approach to malware classification. While this is an exciting area of research, approaches of this nature are typically burdened with false positives rendering the solutions impractical.

This research offers the potential to view executable code in a new light offering insights into two key areas of interests: the nature of viruses, and the executable code itself, offering a new paradigm to the software industry. Software serves a purpose. It has solid attainable goals to suit the end user. Concurrently, it must function within a specified range of parameters to preserve privacy, security, and even safety. The effectiveness of a program under these guidelines can be viewed as its fitness.

Regardless of whether we are analysing virus behavior or developing code that optimizes its fitness, we need a reliable indicator as to its behavior. One of the first successful defenses of malware was to attack the very functions the AM product would call to detect it! This is akin to asking the burglar if he is in your house. Thus behavior detecting components must function at a 'lower' level than those to which the malware may have access. Insuring this can be quite challenging. Nevertheless, if such a framework were in place, the advantages have enormous potential. In terms of viruses and payload detection, we could apply analysis from section 3.4. Equivalently, we could monitor program performance to scan for points of inadequacy. Moreover, in the event of a breach or infection, we could employ the methods of CTI as a means by which the code could heal itself. Again, this necessitates a proper fitness function to evaluate the 'health status' of the executable, and trusted behavioral data from the system.

Of course, there is a wide rift between the musings of theory and the result of implementation, but this framework seems to provide a promising direction for future efforts in this arena.

## 7 Conclusion

To recapitulate, in this paper, we extended the CTI framework presented in [15, 16] to include genetic algorithms and Turing machines. We defined classes of CTI including active, passive and hybrid. Finally, we demonstrated that reproduction emerges as a consequence of the axioms of CTI, both theoretically, and experimentally.

The concepts we presented have great potential for development in future work. As far as the theoretical concepts of this paper are concerned, we have

a lot to do in terms of exploring the potential of passive, active, and hybrid processes. It remains the stance of this paper that hybrid CTI processes are present optimal solutions but this remains to be shown. We would also like to further explore simulation and programming using reproductive nested code. Further, we would like to repeat the experiments of section 4 with different instruction sets, and observe not only the behaviors, but the phenotypes of the solutions in general, to observe overarching patterns among diverse coding alphabets that achieve the same result.

In terms of multidisciplinary studies we have only scratched the surface of the relationship between these concepts and computational molecular biology, computer science, and cyber security.

Further, there is much to do in the way of advancing the theoretical framework of CTI itself. In the next paper, we will visit the ramifications of adding feedback into the intelligence process. Another paper will focus on the global properties of intelligent agent.

## References

- [1] The American Heritage Dictionary. New York: Dell, 2012. Print.
- [2] Bard, Gregory V. "Spelling-error Tolerant, Order-independent Passphrases via the Damerau-Levenshtein String-edit Distance Metric" Proc. of Proceedings of the Fifth Australasian Symposium on ACSW Frontiers. Vol. 69. N.p.: n.p., n.d. 117-24. Print.
- [3] Belshaw, R. "Long-term Reinfection of the Human Genome by Endogenous Retroviruses." Proceedings of the National Academy of Sciences 101.14 (2004): 4894-899. Print.
- [4] Brenner, S., A. O. W. Stretton, and S. Kaplan. "Genetic Code: The 'Nonsense' Triplets for Chain Termination and Their Suppression." Nature 206.4988 (1965): 994-98. Print.
- [5] "Cyber Grand Challenge (CGC)." DARPA RSS. N.p., n.d. Web. 13 Dec. 2013.
- [6] "Defense News & Career Advice ClearanceJobs Defense News & Career Advice." Defense News. N.p., n.d. Web. 13 Dec. 2013.
- [7] Dshalalow, Jewgeni H. Foundations of Real Analysis. [S.l.]: Chapman & Hall (Us), n.d. Print.
- [8] Floreano, Dario, and Claudio Mattiussi. Bio-inspired Artificial Intelligence: Theories, Methods, and Technologies. Cambridge, MA: MIT, 2008. Print.
- [9] Gusfield, Dan. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge [England: Cambridge UP, 1997. Print.



- [10] Hamming, Richard W. "Error Detecting and Error Correcting Codes." Bell System Technical Journal 29.2 (n.d.): 147-60. Print.
- [11] Hopcroft, John E., and Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation. Reading, MA: Addison-Wesley, 1979. Print.
- [12] Jaro, Matthew A. "Probabilistic Linkage of Large Public Health Data Files." Statistics in Medicine 14.5-7 (1995): 491-98. Print.
- [13] Jonoska, Nataša, and Nadrian C. Seeman. DNA Computing :. Berlin: Springer, 2002. Print.
- [14] Koshland, Daniel E., Jr. "The Seven Pillars of Life." Science 295.5563 (2009): 2215-216. Print.
- [15] Kovach, Daniel J., Jr. The Computational Theory of Intelligence: Information Entropy. Proc. of International Conference of Nonlinear Analysis and Applied Mathematics 2013, Greece, Rodos. N.p.: n.p., n.d. Print.
- [16] Kovach, Daniel J., Jr. The Computational Theory of Intelligence: Data Aggregation. Proc. of International Conference of Nonlinear Analysis and Applied Mathematics 2013, Greece, Rodos. N.p.: n.p., n.d. Print.
- [17] Neuhauss, Scott. Handbook for the Deep Ecologist. N.p.: IUiverse, 2005. Print.
- [18] "Payload Definition from PC Magazine Encyclopedia." Payload Definition from PC Magazine Encyclopedia. N.p., n.d. Web. 13 Dec. 2013.
- [19] Snustad, D. Peter., and Michael J. Simmons. Principles of Genetics. Hoboken, NJ: Wiley, 2012. Print.
- [20] Srinivasan, Raghunathan. Protecting Anti-virus Software under Viral Attacks. Thesis. Arizona State University, 2007. N.p.: n.p., n.d. Print.
- [21] Strathern, Paul. Turing and the Computer. New York: Anchor, 1999. Print.